Partial Pathfinding Using Map Abstraction and Refinement

Nathan Sturtevant and Michael Buro
Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{nathanst,mburo}@cs.ualberta.ca

Abstract

Classical search algorithms such as A* or IDA* are useful for computing optimal solutions in a single pass, which can then be executed. But in many domains agents either do not have the time to compute complete plans before acting, or should not spend the time to do so, due to the dynamic nature of the environment. Extensions to A* such as LRTA* address this problem by gradually learning an exact heuristic function, but the learning process is quite slow. In this paper we introduce Partial–Refinement A* (PRA*), which can fully interleave planning and acting through path abstraction and refinement. We demonstrate the effectiveness of PRA* in the domain of real–time strategy (RTS) games. In maps taken from popular RTS games, we show that PRA* is not only able to cleanly interleave planning and execution, but it is also able to do so with only minimal losses of optimality.

Introduction and Related Work

Consider the problem of driving a car from Los Angeles to New York. A human approaching this task would likely begin by first answering high-level questions, such as which states to drive through. But low-level decisions such as which lane on the highway to drive in will not even be considered until moments before it is necessary. Furthermore, if we take a short detour around traffic in Tulsa, we will not have to revise any computed plans about what to do after leaving Tusla. Similarly, if we change our final destination in New York, we do not have to re-plan our entire route; just the last few steps. In fact, we do not even have to consider these details until we arrive in New York. This planning involves several levels of reasoning. First, it requires an abstract model of the world, so we can reason at both low levels (what lane to drive in) as well as high levels (what cities to visit en route). It also involves planning and executing partial plans or paths through the world. It would be unreasonable to consider planning every lane change for the entire trip before setting out. Yet, this is exactly how traditional search algorithms have approached the task, building a complete plan before starting. In this paper, we introduce the Path-Refinement A* (PRA*) algorithm which can build high-level plans about the world, and progressively refine them into low-level actions as needed. Partial-path refinement means building paths in a manner that interleaves acting and planning, and thus spreading cost of path computation more evenly over the path execution time. This is a highly desirable property, providing robustness in the face of a dynamic environment and minimizing the amount of recomputation that needs to be done when the world changes.

One application we are particularly interested in is real-time strategy (RTS) games. RTS games make up a significant portion of the computer game market; titles such as Starcraft and Warcraft by Blizzard Entertainment have sold millions of copies. RTS players are assisted by the computer in tasks such as pathfinding, yet pathfinding also tends to be one of the most criticized parts of many games, because existing pathfinding systems can easily be confused. Additionally, the real-time graphics and simulation demands of RTS games leave only a small portion of the CPU for AI tasks, meaning that pathfinding must be extremely efficient. RTS games can also be viewed as abstract simulations in which robots move around and interact. Therefore, robot navigation can also benefit from pathfinding algorithms which can interleave pathplanning and plan execution.

Related Work

A* (Hart, Nilsson, & Raphael 1968) and IDA* (Korf 1985) have been explored thoroughly with regard to finding optimal paths in a well-known and stationary environment. D*-Lite (Koenig & Likhachev 2002) is able to do limited replanning if the world changes, but it can't handle changes like a moving target. Furthermore, these algorithms are not necessarily well-suited for acting in dynamic environments, because they cannot fully interleave planning and acting. LRTA* (Korf 1990), which learns a perfect heuristic function, is more suited for this, but it can take a significant amount of time for LRTA* to learn an accurate heuristic function. A comparison of D*-lite and LRTA* can be found in (Koenig 2004). LRTA* is shown to work well in environments where there are only minor obstacles between the start and goal, which means the heuristic is very accurate to begin with. But, in domains where the initial heuristic is poor, LRTA* also performs quite poorly. Thus, if we are in a non-stationary environment, do not have a good initial-heuristic (cannot guarantee heuristic quality is high), or we do not have time to do full planning before starting to act, we need something better than these algorithms.

While the ideas of abstraction and path refinement are not new, they have not previously been developed and analyzed with regard to partial solution generation. There is, however, a wide body of work related to speeding up A* search. (Botea, Müller, & Schaeffer 2004) does an excellent job of reviewing research on pathfinding, and (Reese & Stout 1999) also contains an overview of different met-

rics that we may want to optimize during search. Notably absent from the overview is partial pathfinding. Similar to our work, (Holte *et al.* 1996a) uses abstraction and refinement in a graph representation of the problem space to speed solution time. Abstraction and refinement can also be used to build heuristic functions for unabstracted space (Holte *et al.* 1996b). Instead of abstracting directly from a graph representation, Hierarchical Pathfinding A* (HPA*) (Botea, Müller, & Schaeffer 2004) overlays a map with large sectors and calculates optimal paths between limited sets of entrances and exits to the sectors. Through the process of smoothing, high quality paths can be obtained. Beyond simple pathfinding, abstraction has been investigated in other areas such as robotics (Fernandez. & Gonzalez 2001) and planning (Yang, Tenenberg, & Woods 1996).

The remainder of the paper is organized as follows: we first present the spatial abstraction technique we are using and the partial pathfinding algorithm. Then, we show empirical performance results of PRA* in various settings, and conclude by outlining future work.

Abstractions

In order for a search algorithm that only calculates partial solutions to be complete, it must have some higher-level information about the structure of the world. For instance, if we try to use A* for partial planning by limiting the search to some depth d, we are vulnerable to being trapped in a dead-end of size $d + 1$, unless the terrain is pre-defined in a way that guarantees we can avoid such traps. Instead of relying on specific problem features, we instead rely on the abstractability of the problem space. In pathfinding, for instance, we can treat four neighboring tiles as a single abstract tile located at the center of the original tiles. By building an abstraction hierarchy of the world, any problem can be quickly solved in abstract space. If we then use the abstracted solution as a guide for generating a solution in the actual problem space, we are guaranteed to never get trapped in local dead-ends, and to generate feasible solutions. We will discuss the full PRA* algorithm that uses abstractions in the next section, but first we will cover how abstractions can be built and used.

Automatically Building Abstractions

A search problem is traditionally defined as a tuple, $\{\mathcal{S}, s, \mathcal{G}, \mathcal{O}, \mathcal{H}\}$, where $\mathcal{S}$ is the set of all states, s is a start state, $\mathcal{G}$ is the set of goal states or a goal test function, $\mathcal{O}$ is a set of operators, and $\mathcal{H}$ is a heuristic function. In order to build an abstracted space from a search problem, we need to enhance this problem representation. A search problem can be described as or transformed into a graph, where nodes are states and edges are operators. An abstract graph is a reduction of the full state graph, where each node represents one or more states in the lower level graph, and an edge exists between two nodes if there is any operator which can be applied to any state abstracted by the first node which will take you into any other state abstracted by the second node. We define an *abstractable search problem* as the tuple $\{\mathcal{S}^0, s, \mathcal{G}, \mathcal{O}, \mathcal{H}^k, \mathcal{A}\}$. The abstractable search problem

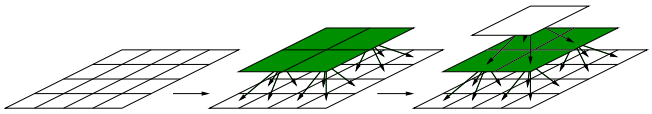

Figure 1: Abstracting tiles into a hierarchy

is enhanced from a standard search problem in the following ways: a state is now associated with an abstraction level, so $\mathcal{S}^k$ is the set of states that has been abstracted at the k^{th} level. $\mathcal{S}^0$ must be defined as part of the problem, but all other levels of abstraction can be built given an abstraction function $\mathcal{A}(s_1^k ... s_i^k) \rightarrow s_j^{k+1}$, which takes i states at the k^{th} level of abstraction and maps them to a single state at the abstraction level $k + 1$. Each node is only abstracted once. Finally, the heuristic function must be enhanced to take two states at any level of abstraction, k, and return an estimate of their distance. Given an abstractable search problem we can automatically build abstractions of $\mathcal{S}^0$ using a variety of methods. While an abstraction function may be able to take any arbitrary set of states and abstract them, in practice we generally want to abstract states that are local to each other.

Abstractions for Pathfinding

For pathfinding, the world is commonly discretized into tiles, which we then abstract based on their octile connectivity; that is their connectivity with respect to the eight immediately adjacent tiles. A simple example for an empty map is shown in Figure 1. Abstraction can be seen as the process of reducing the resolution of the map, while maintaining connectivity information in the abstraction. In an empty map, a grid of sixteen tiles is quite similar to an abstract grid of just four tiles, which again can be represented by a single tile. In practice we only use the tile representation at the bottom level of the hierarchy, and use a graph representation at higher levels of abstraction. Instead of overlaying some structure upon the search space — like in HPA* — we instead abstract states based on local features of states. We abstract nodes based on two patterns: *cliques* and *orphans*. The largest clique we look for in this domain is a 4-clique. When building static abstractions, we first iterate through the space looking for cliques in the nodes that have yet to be abstracted. These cliques are reduced to a single node in the parent abstraction. By abstracting cliques we are guaranteed that all nodes in an abstraction are able to reach any other state in the abstract node within a single step, except *orphaned* nodes. An orphan is a node that can only be reached by a single operator. When abstracting the graph, we just join them into the same abstracted node as their neighbor.

We demonstrate these methods in Figure 2. In this example we start with a graph containing twelve nodes and 18 edges. There are two cliques of (maximal) size 4, so those are reduced into sets A and B. Connected to the clique in set A there is also an orphan that can only be reached through that set, so it is merged into A as well. There are single nodes connecting sets A and B. Node D has no neighbors, so it remains alone, but node C has a single orphan, which also forms a 2-clique, so these nodes are abstracted together. The final abstracted graph has four nodes and

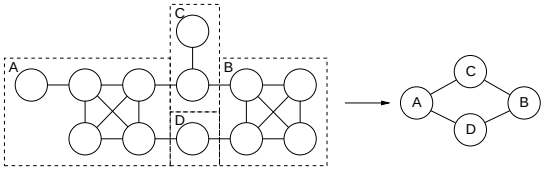

Figure 2: Abstracting a general graph

four edges. While we do not show the process explicitly, we could repeat the reduction twice more, first creating a graph with two nodes and one edge, and then reducing the graph to a single node. In our pathfinding domain, given an abstracted node, we define the abstracted position to be the average position of all nodes abstracted by that graph. The heuristic distance between any two nodes is then the octile distance between the abstract location of those nodes, $\sqrt{2} \cdot \min(|\Delta x|, |\Delta y|) + | \, |\Delta x| - |\Delta y| \, |$.

Abstraction Costs

Given n nodes at the bottom level of an abstraction hierarchy, the expected height of the hierarchy will be $O(\log n)$. There are pathological orderings that will create a hierarchy of height $\Theta(n)$, but they do not occur in practice. To abstract a single graph g, we will visit each node in g at most a constant number of times, assuming the graph is sparse. So, the total time to build the hierarchy will be $O(n)$. If our knowledge of the world is incomplete or the world topology is dynamic, it is possible to repair abstractions in $O(\log n)$ time per update. A full description of these methods, however, is beyond the scope of this paper.

Pathing Through Refinement

In this section we demonstrate a simple method, *QuickPath*, for finding plausible paths in the world without doing any significant search. After building an abstraction hierarchy, all connected nodes will be abstracted into a single node at the highest level of abstraction. Thus, to check if any two nodes are connected, we can simply check to see if they ever merge into the same parent within the abstraction hierarchy. If the base-level graph has n nodes, this will take $O(\log n)$ time. This can be used as a quick check for pathability between two nodes, but can also be expanded for generating full paths, which we demonstrate in Figure 3. In this figure, we wish to find a path between A and D. In practice there will be many more nodes at each level, but we only show the nodes relevant to this example. As a first step, we simply traverse the hierarchy looking for a common parent for A and D. We find this in abstraction level 2 at node $ABCD$. Because AB and CD are part of the same abstract

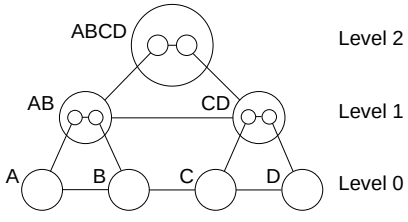

Figure 3: Using abstraction to quickly refine a path

PRA* ($abstractGraph$, $start$, $goal$, k)
  GetAbstractionHierarchy ($start$, $goal$)
  s = **GetStartLevel** ($start$, $goal$)
  empty $path$
  for each level i=s..1
    $path$ = **RefinePath** ($path$, $start[i]$, tail($path$))
    truncate $path$ to length k
  return $path$

RefinePath ($path$, $start$, $goal$)
  return **aStar** ($start$, $goal$) subject to nodes in $path$

Figure 4: PRA*(k) Pseudo-Code

node and not orphans, they must have an edge between each other at abstraction level 1. We are then faced with the general problem of refining the path. There are two components of a path that must be refined in order to create a path at a lower level of abstraction, nodes and edges. To refine the edge between AB and CD we must look at the nodes in AB and see which of them has an edge connecting to any node in CD. In this case, node B in AB connects to node C in CD. Then, we must refine the portion of the path that passes through nodes AB and CD. But, because each node is made of cliques, this is trivial, since we are guaranteed by our abstraction mechanism that there is an edge between A and B, and similarly between C and D.

Function *QuickPath* returns a path between any two nodes as described above. In the worst case, the total time to build this path will be $O(\sum_{i=1}^{\log n} |p_i|)$, where p_i is the refined path at level i. *QuickPath* will find *a* path between two nodes, but the path generated could be highly suboptimal. Still, we use this method as a stepping stone to introduce Partial-Refinement A*, which refines this simple method to produce higher-quality results.

Partial-Refinement A*

We show the pseudo-code for Partial-Refinement A* (PRA*) in Figure 4. PRA* works similarly to *QuickPath*, but it contains four enhancements which we describe here. First, it uses a heuristic to guide search at any level of abstraction, instead of just refining the nodes on a path. Second, it allows path refinement to occur in a corridor or swath outside of the abstract path. Third, it does not start planning from the top of the abstraction hierarchy. Finally, it only does partial pathfinding at each step.

Replacing Refinement with Search

The *QuickPath* algorithm does not perform search like classical A*. There may be multiple ways of doing refinement of a path, but it makes no effort to find better ways of refinement. Instead of generating any candidate path between nodes as *QuickPath* does, we instead use a single A* search through an abstracted level, using the abstract heuristic function $\mathcal{H}^k$. We minimize the cost of search by only allowing A* to generate nodes whose parents are part of the abstract path we are refining. So, each abstract path defines a swath of nodes in a lower level of abstraction, through which A* finds an optimal path. However, finding an optimal path

through this swath does not guarantee global optimality.

Choosing Abstraction Levels

One explanation for the sub-optimality is that we are beginning our pathfinding at a level that is too abstract. If we instead begin on the finest grid, we would be able to find an optimal path. Thus, we would like to dynamically find an abstraction level that is neither too coarse to introduce significant sub-optimality nor too fine to introduce significant search costs. In the previous section we demonstrated how traversing the parents of any two nodes will determine the pathability between those nodes. In the worst case this operation will take $O(\log n)$ time, but, if we assume that any given node has an equal probability of being merged with any of its neighbors in any abstraction step, then the expected level at which two nodes are merged is $\Theta(\log |p|)$, where p is the path between those nodes. So, the process of traversing the parents of two nodes until they merge is not only a quick check for pathability, it is also a very coarse heuristic measure of the path length between two nodes. Additionally, in any abstraction hierarchy there will always be at least two nodes who are adjacent in the original problem graph, which do not merge until the last step of abstraction. So, we can increase the accuracy of this heuristic by measuring the level in the abstraction at which two nodes are first connected by a single edge. We currently use this heuristic to choose to select the level at which we begin our initial path, as we can balance between too granular a starting level, which leads to suboptimal results, and too fine a starting level, which leads to slower performance. We opt for the level of abstraction half-way between where two nodes are first connected by an edge in the abstraction hierarchy, but it is a point of future research to investigate this in more detail.

Partial Refi nement

The final enhancement of PRA* over *QuickPath* is that instead of refining the entire abstract path from beginning to end, we instead only refine a partial segment of the abstract path at each step. We do this by truncating the path we are refining to a fixed length, and then searching for an optimal path to the last node in this truncated path. We also exit as soon as we find an optimal path to any node that is abstracted by the goal state. Because we always have a complete path from the start to goal state at some level of abstraction, we can guarantee that we will eventually reach our goal state. As an additional optimization, we can cache some of the high-level planning that is done in partial refinement to make the process as efficient as PRA* with full refinement. The version of PRA* which does not do partial refinement, we refer to as PRA*(∞). Otherwise, PRA*(k) refers to the fact that we are refining k nodes out of each abstract path.

Experiments

In this section we present empirical performance results of PRA* when neglecting the time it takes to build map abstractions. In a future paper we will analyse the performance of a dynamic PRA* variant applied to dynamic environments. Here, we will first demonstrate that PRA*(∞) has similar performance properties as HPA*: it is much faster than A* in static pathfinding setups — while its solution quality is very close to optimal with high probability. In a second set of experiments we will measure the PRA*(k) partial pathfinding performance on the complete path problem and when planning and execution can be interleaved.

In a first phase, we extracted maps from the popular games Baldur's Gate and Warcraft 3, discarding those that were either too small or consisted of only one large connected area. We then scaled up the resulting 116 maps to size 512×512 while retaining their topological structure. In a final preprocessing step, we generated a large number of random location pairs and saved ten optimal paths for each map and every path bucket i between 0 and 127 (a path of length l belongs to bucket i if and only if $i = \lfloor l/4 \rfloor$). This procedure generated 1160 paths in each bucket of the 128 buckets we considered totaling 148480 paths with length between 0 and 511. Maps are represented as square tile grids with the commonly used octile neighborhood relation. Blocked tiles are marked as such. At any given time during pathfinding the moving object is located on an unmarked tile and is allowed to proceed to one of its up to eight unmarked neighbors. We are not allowed to cut corners diagonally, i.e. in case of three unmarked tiles in a 2×2 block, diagonal moves through the center are not possible. All experiments are single-threaded and were run on a dual-CPU Power-Mac running at 2 GHz with 1 GB of RAM using gcc 3.3.

PRA*(∞) Complete Path Performance

The first set of experiments examines how PRA*(∞) performs on the classic pathfinding problem — namely to find a shortest path between two locations on a static map — if one exists. We use A* as a benchmark algorithm to judge the quality of PRA*(∞) paths and its runtime. Neither A* nor PRA* have been fully optimized for speed. Our A* implementation, however, utilizes a heap for the open list and a look-up table for the closed list. Figures 5a) and b) show interesting percentiles of the A* and PRA*(∞) runtime with respect to the (optimal) A* path length, which we use as a simple path complexity measure. Fig. 5a) demonstrates that A* for increased path length eventually explores large fractions of the search space which here is roughly of size $512^2/2$. The median (50% percentile) not being centered between the 5$^{\text{th}}$ and 95$^{\text{th}}$ percentiles indicates that A*'s runtime distribution is skewed, meaning the mean and variance are not adequate to describe the results. PRA*'s runtime grows more slowly with respect to the path length, which is what was to be expected from the analyses in the previous sections. Its variance is also much smaller. We also measured runtime in terms of expanding nodes in both algorithms, which also is the number of calls to the h function. Both measures are highly correlated. On the computer we used for the experiments one micro second of execution time on average corresponds to 2.2 expanded nodes. For a more detailed runtime comparison, we have plotted percentiles for runtime ratios based on individual paths in Fig. 5c). Again, the result is in favour of PRA*(∞), whose gains appear to increase approximately linearly with longer paths. However, only looking at the runtime does not convey the entire picture because PRA* computes approximate paths. Fig. 6a)

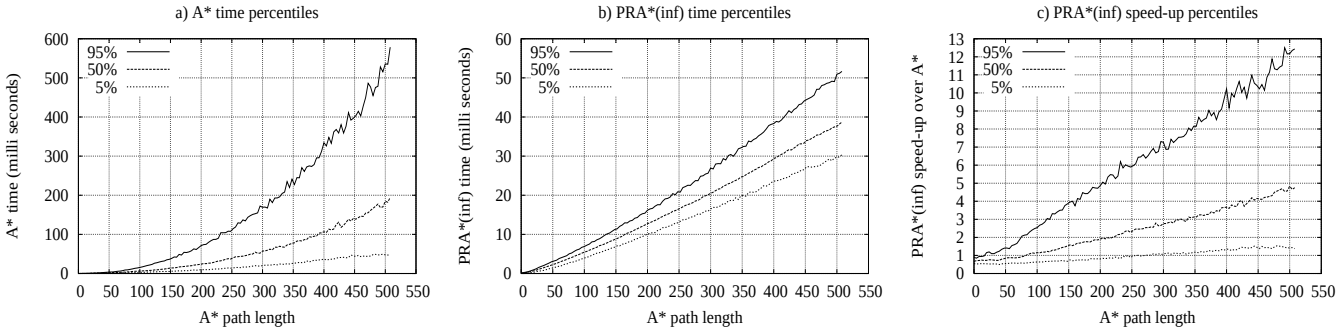

Figure 5: Runtime performance of A* and PRA*(∞)

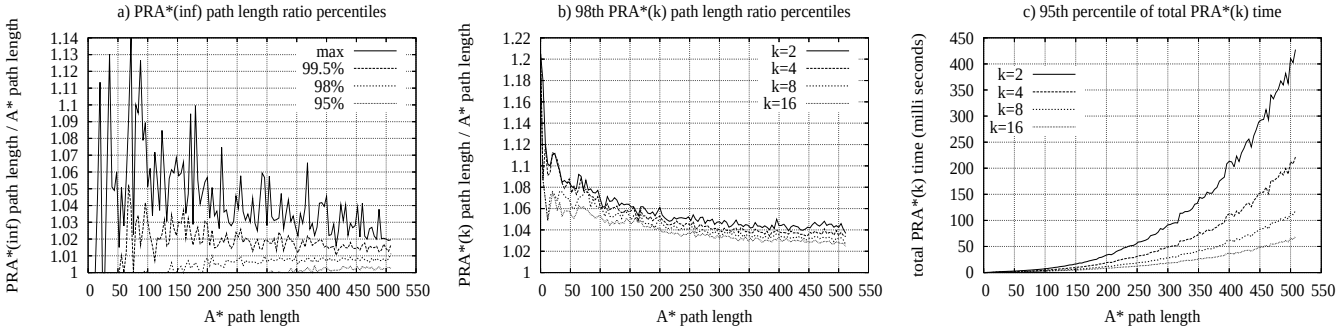

Figure 6: PRA*(∞) and PRA*(k) path quality, total PRA*(k) runtime

shows that PRA*(∞) is very accurate with high probability. For instance, we can see that 98% of the time the PRA*(∞) path length is within 1% of optimal, and 95% of the time it is better than 1.005 $\times$ optimal. Looking at the maximal ratio values, it appears that PRA*(∞) paths more than 10% above optimal are very rare. (Botea, Müller, & Schaeffer 2004) report similar performance – maximum 10 times speed up and less than 1% error from optimal.

PRA*(k) Complete Path Performance

In the following experiment we determined the quality of complete paths constructed by PRA*(k) for $k = 2, 4, 8, 16$. Fig. 6b) shows the 98$^{\text{th}}$ percentile of the path length ratio when computing the full path with PRA*(k). The paths are longer compared with PRA*(∞), but still short enough to be acceptable for many purposes. The total runtime of PRA*(k) is super-linear in the path length (Fig. 6c), which is caused by the increasing number of planning steps ($\approx C{\cdot}\text{length}/k$), the longer time for top-level searches, and more levels to be searched. Increasing k reduces the total runtime considerably, but it would increase the cost of individual planning steps and thus decrease performance if replanning was triggered by external events such as changing the target location. The total runtime of PRA*(16) for paths of length 508–512 is roughly equal to the PRA*(∞) time ($\leq$ 60 msec in 95% of the cases). The difference is that PRA*(k) can spread this computation time over the entire path, which we consider next.

Interleaving Path Planning and Excecution

To measure PRA*'s performance in environments which permit simultaneous pathfinding and execution, we model the timing conditions in RTS games. Because RTS games — which at their core are clocked simulations — often feature a large number of moving objects and large terrains, pathfinding is currently one of the most time-consuming RTS AI tasks. In each simulation frame, game objects can receive instructions from players or AI modules, which then get executed. In the following timing model we account for the fact that object motion and path planning can be interleaved. Assuming a static environment, the total cost in simulation cycles, t_{sim}, to move a unit to a goal location has three components: 1) the initial planning time, 2) the sum of maxima of the planning time for the next step and the execution time for the previous step, and 3) the time to execute the last step in the world. To formalize this definition, t_i ($1 \leq i \leq n$) is the time used for planning move sequence i of Euclidean length l_i measured in tile-widths, t_f is the simulation frame period, t_m is the maximal time in a frame that can be used for pathfinding for a single object, and s is the speed of the object (tile-widths per second). Thus,

$$t_{\text{sim}} = \lceil \frac{t_1}{t_m} \rceil + \sum_{i=2}^{n} \max(\lceil \frac{t_i}{t_m} \rceil, \lceil \frac{l_{i-1}}{s \cdot t_f} \rceil) + \lceil \frac{l_n}{s \cdot t_f} \rceil \quad (1)$$

Note that this timing model is quite general and also applies to "real" robot motion planning.

For the experiments we chose $t_f = 100$ msec (i.e. 10 frames per second) which is typical for RTS games. A small t_m value forces the pathfinding system to spread path planning over multiple simulation cycles, if more time than t_m is required. This technique — which is commonly used in RTS games in conjunction with regular A* — limits the load of the CPU caused by pathfinding and frees it up for other time consuming tasks such as graphics. As a baseline we com-

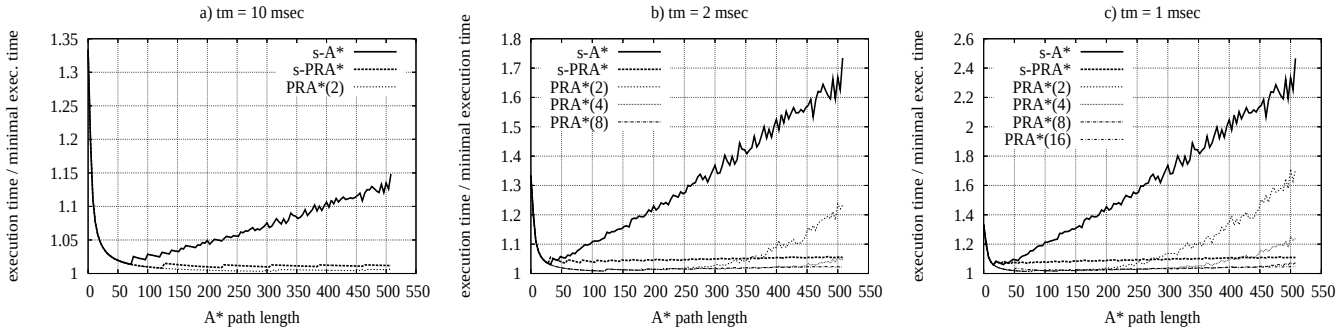

Figure 7: 98th percentile curves of path execution slowdowns in clocked RTS game simulations for $t_m = 10, 2$, and 1 msec

pare PRA*(k) to versions of A* and PRA*(∞), which we call s-A* and s-PRA*, that do all their planning in the first step which gets spread over multiple frames if necessary.

Unit speed s influences the total path execution time in equation (1) directly: the faster units are, the more thinking time contributes to t_{sim}. The way popular RTS games are designed, the choice of $s = 10$ tiles/sec is quite high and leading us to more conservative conclusions. In Fig. 7 the simulation timing results for s-A*, s-PRA*, and PRA*(k) with caching are summarized in form of 98th percentile curves. Here we consider the ratio of t_{sim} and the minimal possible path execution time in the clocked simulation, i.e. disregarding thinking time in (1). The spread-execution variants of A* and PRA*(∞) plan only once per path, so n is set to 1. As seen in all graphs in Fig. 7, for short paths the initial planning step outweights the path execution time resulting in a singularity-like relation around 0. For longer paths spreading the path computation does not help s-A* or s-PRA* anymore because the runtime of A* and PRA*(∞) is superlinear in the path length (Fig. 5 a,b). It is also apparent that frequent partial refinement comes at a price, especially when the paths are long. However, choosing $k = 16$ results in an excellent real-time performance for the large maps we considered. The path execution time is better than for s-A* and s-PRA* and replanning triggered by external events will not slow PRA*(k) down as much as s-PRA*. Even if the planning time is restricted to just 1% (1 msec/frame), long paths generated by PRA*(16) are less than 5% away from optimal in 98% of the cases. The remaining CPU time (99%) is available for other game tasks including simultaneously finding paths for more objects.

Conclusions and Future Work

In this article we have presented an A* variant, Path-Refinement A*, which speeds up pathfinding both in complete path generation and in partial-path generation, at the cost of a slight decrease in path quality. This is the first time that partial pathfinding methods have been implemented and analyzed. Our results show that partial refinement is quite effective at interleaving planning and acting, and thus can free up CPU time for other tasks. Our current implementation treats all other agents in the world as static obstacles to be avoided. Given the success of PRA*, we would like to extend our implementation to allow cooperative behavior between agents in the world, as well as investigating other scenarios, such as chasing games. Other topics of interest are establishing tight theoretical average and worst case time and quality bounds for PRA*, improving the computation of the start level and refinement parameter k, and applying PRA* to other planning domains.

Acknowledgments

We thank Markus Enzenberger for valuable feedback on this paper. Financial support was provided by NSERC and Alberta's Informatics Circle of Research Excellence (iCORE).

References

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Near optimal hierarchical path-finding. *J. of Game Develop.* 1(1):7–28.

Fernandez., A., and Gonzalez, J. 2001. *Multi-Hierarchical Representation of Large-Scale Space*. Kluwer.

Hart, P.; Nilsson, N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. on Systems Science and Cybern.* 4:100–107.

Holte, R.; Mkadmi, T.; Zimmer, R. M.; and MacDonald, A. J. 1996a. Speeding up problem solving by abstraction: A graph oriented approach. *Artif. Intell.* 85(1–2):321–361.

Holte, R.; Perez, M.; Zimmer, R.; and MacDonald, A. 1996b. Hierarchical A*: Searching abstraction hierarchies efficiently. In *AAAI/IAAI Vol. 1*, 530–535.

Koenig, S., and Likhachev, M. 2002. Improved fast replanning for robot navigation in unknown terrain. *http://citeseer.ist.psu.edu/koenig02improved.html*.

Koenig, S. 2004. A comparison of fast search methods for real-time situated agents. In *Proceedings of the Third International Joint Conference on Autonomous Agents and MultiAgent Systems*, 864–871. ACM.

Korf, R. 1985. Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intelligence* 27(1):97–109.

Korf, R. 1990. Real-time heuristic search. *Artificial Intelligence* 42(2-3):189–211.

Reese, B., and Stout, B. 1999. Finding a pathfinder http://citeseer.ist.psu.edu/reese99finding.html.

Yang, Q.; Tenenberg, J.; and Woods, S. 1996. On the implementation and evaluation of ABTweak. *Computational Intelligence Journal* 12(2):295–318.